Slim

中文文档

目錄

Slim 中文文档

> 来源：Slim 中文文档

此处的文档针对 **Slim 3** 。要查阅 Slim 2 的文档请访问 docs.slimframework.com 。

欢迎使用

Slim是一款 PHP 微框架，可以帮助你快速编写简单但功能强大的 web 应用和 API 。在它的核心，Slim 是一个调度程序，它接收一个 HTTP 请求，调用一个适当的回调例程，然后返回一个 HTTP 响应。就这样简单。

重点是什么？

Slim 是一个理想的工具，用来创建销毁、重用或发布数据。Slim 也是一个用来快速构建原型的好工具。呃，那啥，你甚至可以用 Slim 来创建带有用户界面的功能完整的 web 应用程序。更重要的是，Slim 速度超快，代码很简单。实际上，一个下午你就能读完并理解它的源代码！

> 在它的核心，Slim 是一个调度程序，它接收一个 HTTP 请求，调用一个适当的回调例程，然后返回一个 HTTP 响应。就这样简单。

你并不总是需要一套全方位的解决方案，比如 Symfony 或者 Laravel 。当然，它们都是了不起的工具。但它们往往矫枉过正。相反，Slim 只最小限度地提供你所需要用的工具，没有其他多余的东西。

它如何工作？

首先，你需要一个 web 服务器，比如 Nginx 或 Apache 。你需要配置你的 web 服务器让它发送所有适当的请求到一个“前端控制器” PHP 文件。你就在这个 PHP 文件里实例化并运行你的 Slim 应用。

一个 Slim 应用包含多个响应特定 HTTP 请求的路由。每个路由调用一个回调并返回一个 HTTP 响应。在刚开始的时候，你首先实例化并配置 Slim 应用。下一步，定义你的应用程序的路由。最后，运行你的 Slim 应用程序。这很简单吧？这里有一个实例应用程序：

```
<?php
// 创建并配置 Slim app $app = new \Slim\App;

// 定义 app 路由 $app->get('/hello/{name}', function ($request, $response, $args) {
    return $response->write("Hello " . $args['name']);
});

// 运行 app $app->run();
```

Figure 1: Slim 示例应用

请求和响应

当你构建一个 Slim 应用时，你会经常直接与请求对象和响应对象打交道。这些对象表示由 web 服务器实际接收到的 HTTP 请求，以及最终返回给客户端的 HTTP 响应。

Slim 应用程序的每个路由都将前的请求对象和响应对象作为它的回调例程的参数。这些对象实现了流行的 PSR 7 接口。Slim 应用的路由可以必要地检查或操作这些对象。最终，每个 Slim应用路由必然返回一个 PSR 7 响应对象。

加入你自己的组件

Slim 可以良好地兼容其他 PHP 组件。你可以注册第三方组件，比如 Slim-Csrf, Slim-HttpCache, 或 Slim-Flash 这种基于 Slim 默认功能上的组件。Slim 还可以很容易地集成那些从Packagist上找到的第三方组件。

如何阅读这份文档

如果你是第一次接触 Slim，我建议你从头到尾先阅读一遍。如果你已经和 Slim 混得比较熟了，你可以直接去阅读你想看的部分。

本文档从解释 Slim 的概念和架构开始，而不是贸然开始讲述请求和响应处理、路由、错误处理这些特定的话题。特定的话题。（中文文档若有错误或不准确，请务必反馈一下哦）

安装

系统要求

- 支持 URL 重写的 Web 服务器
- PHP 5.5.0 或者更新版本

如何安装 Slim

我们推荐你使用 Composer 。这个依赖管理器来安装 Slim 框架。在你的项目根目录中，运行以下 bash 命令，将最新的稳定版安装到项目的 `vendor/` 目录中。

```
composer require slim/slim "^3.0"
```

然后，在 PHP 脚本中载入 Composer 的自动加载器，然后就可以开始使用 Slim 了。

```
<?php
require 'vendor/autoload.php';
```

如何安装 Composer

没有 Composer ？很容易安装的。由于GFW的原因，所以建议使用国内镜像来下载安装。

通过 Composer 镜像安装 composer

首先，务必确保已经正确安装了 PHP。

局部安装

局部安装是将 composer 安装到当前目录下面（比如安装到项目根目录下），然后就可以通过 php composer.phar 来使用 composer 了。

Mac 或 Linux 系统：打开命令行窗口并执行如下命令：

```
curl -sS http://install.phpcomposer.com/installer | php
```

Windows 系统（Mac 或 Linux 系统也可以使用）：请执行如下命令：

```
php -r "readfile('http://install.phpcomposer.com/installer');" | php
```

升级指南

如果你打算从 Slim 2 升级到 Slim 3，这里有一些重要的变化，你必须清楚。

新的 PHP 版本要求

Slim 3 要求 PHP 5.5+

新的路由函数签名

```
$app->get('/', function (Request $req,  Response $res, $args = []) {
    return $res->withStatus(400)->write('Bad Request');
});
```

获取 _GET 和 _POST 变量

```
$app->get('/', function (Request $req,  Response $res, $args = []) {
    $myvar1 = $req->getParam('myvar'); //检查 _GET 和 _POST [不遵循 PSR 7]
    $myvar2 = $req->getParsedBody()['myvar']; //检查 _POST  [遵循 PSR 7]
    $myvar3 = $req->getQueryParams()['myvar']; //检查 _GET [遵循 PSR 7]
});
```

钩子 / Hooks

Slim v3 不再有钩子的概念。You should consider reimplementing any functionality associated with the default hooks in Slim v2 as middleware instead. If you need the ability to apply custom hooks at arbitrary points in your code (for example, within a route), you should consider a third-party package such as Symfony's EventDispatcher or Zend Framework's EventManager.

移除 HTTP 缓存

在 Slim v3 我们将 HTTP 缓存迁移到了单独的模块中： Slim\Http\Cache.

移除 Stop/Halt

Slim Core has removed Stop/Halt. In your applications, you should transition to using the withStatus() and withBody() methods.

重定向的改变

在 Slim v2.x 我们需要使用助手函数 $app->redirect(); 来触发重定向请求。在 Slim v3.x 中，我们可以使用响应类来做这事。

Example:

```
$app->get('/', function ($req, $res, $args) {
  return $res->withStatus(302)->withHeader('Location', 'your-new-uri');
});
```

中间件签名

中间件的签名已经从一个类变成了函数。

新的签名：

```
use Psr\Http\Message\RequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

$app->add(function (Request $req,  Response $res, callable $next) {
    // Do stuff before passing along
    $newResponse = $next($req, $res);
    // Do stuff after route is rendered
    return $newResponse; // continue
});
```

你仍然可以使用类来做签名：

```
namespace My;

use Psr\Http\Message\RequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

class Middleware
{
    function __invoke(Request $req,  Response $res, callable $next) {
        // Do stuff before passing along
        $newResponse = $next($req, $res);
        // Do stuff after route is rendered
        return $newResponse; // continue
    }
}

// Register
$app->add(new My\Middleware());
// or
$app->add(My\Middleware::class);
```

Middleware Execution

Application middleware is executed as Last In First Executed (LIFE).

Flash Messages

Flash messages are no longer a part of the Slim v3 core but instead have been moved to seperate Slim Flash package.

Cookies

In v3.0 cookies has been removed from core. See FIG Cookies for a PSR-7 compatible cookie component.

Removal of Crypto

In v3.0 we have removed the dependency for crypto in core.

New Router

Slim now utilizes FastRoute, a new, more powerful router!

This means that the specification of route patterns has changed with named parameters now in braces and square brackets used for optional segments:

```
// named parameter:
$app->get('/hello/{name}', /*...*/);

// optional segment:
$app->get('/news[/{year}]', /*...*/);
```

Route Middleware

The syntax for adding route middleware has changed slightly. In v3.0:

```
$app->get(…)->add($mw2)->add($mw1);
```

urlFor() is now pathFor() in the router

`urlFor()` has been renamed `pathFor()` and can be found in the `router` object:

```
$app->get('/', function ($request, $response, $args) {
    $url = $this->router->pathFor('home');
    $response->write("<a href='$url'>Home</a>");
    return $response;
})->setName('home');
```

Also, `pathFor()` is base path aware.

Container and DI … Constructing

Slim uses Pimple as a Dependency Injection Container.

```
// index.php
$app = new Slim\App(
    new \Slim\Container(
        include '../config/container.config.php'
    )
);

// Slim will grab the Home class from the container defined below and execute its index m
// If the class is not defined in the container Slim will still contruct it and pass the
$app->get('/', Home::class . ':index');

// In container.config.php
// We are using the SlimTwig here
return [
    'settings' => [
        'viewTemplatesDirectory' => '../templates',
    ],
    'twig' => [
        'title' => '',
        'description' => '',
        'author' => ''
    ],
    'view' => function ($c) {
        $view = new Twig(
            $c['settings']['viewTemplatesDirectory'],
            [
                'cache' => false // '../cache'
            ]
        );

        // Instantiate and add Slim specific extension
        $view->addExtension(
            new TwigExtension(
                $c['router'],
                $c['request']->getUri()
            )
        );

        foreach ($c['twig'] as $name => $value) {
            $view->getEnvironment()->addGlobal($name, $value);
        }

        return $view;
    },
    Home::class => function ($c) {
        return new Home($c['view']);
    }
];
```

PSR-7 Objects

Request, Response, Uri & UploadFile are immutable.

This means that when you change one of these objects, the old instance is not updated.

```
// This is WRONG. The change will not pass through.
$app->add(function (Request $request, Response $response, $next) {
    $request->withAttribute('abc', 'def');
    return $next($request, $response);
});

// This is correct.
$app->add(function (Request $request, Response $response, $next) {
    $request = $request->withAttribute('abc', 'def');
    return $next($request, $response);
});
```

Message bodies are streams

```
// ...
$image = __DIR__ . '/huge_photo.jpg';
$body = new Stream($image);
$response = (new Response())
     ->withStatus(200, 'OK')
     ->withHeader('Content-Type', 'image/jpeg')
     ->withHeader('Content-Length', filesize($image))
     ->withBody($body);
// ...
```

For text:

```
// ...
$response = (new Response())->getBody()->write('Hello world!')

// Or Slim specific: Not PSR-7 compliant.
$response = (new Response())->write('Hello world!');
// ...
```

Web 服务器

典型地使用前端控制器将 web 服务器接收到的 HTTP 请求汇集到一个单独的 PHP 文件。下面的文档说明了如何通知你的 web 服务器将接收到的 HTTP 请求发送到 PHP 前端控制器文件。

PHP built-in server

Run the following command in terminal to start localhost web server, assuming `./public/` is public-accessible directory with `index.php` file:

```
php -S localhost:8080 -t ./public/
```

Apache 配置

确保 `.htaccess` 和 `index.php` 这两个文件位于同一个可公开访问的目录中。这个 `.htaccess` 文件应当包含以下代码：

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ index.php [QSA,L]
```

为了保证 `.htaccess` 的重写（rewrite）规则能正常生效，务必确保你的 Apache 虚拟主机开启了 `AllowOverride` 选项。

```
AllowOverride All
```

Nginx 配置

这是一个例子，在 Nginx 虚拟主机上针对域名 `example.com` 的配置。它监听80端口上的入境（inbound）HTTP 连接。它假定一个PHP-FPM服务器在端口9000上运行。你需要将 `server_name` , `error_log` , `access_log` , 和 `root` 这些指令修改成你自己的值。其中 `root` 指令是你的应用程序公共文件根目录的路径；你的 Slim 应用的 `index.php` 前端控制器文件应该放在这个目录中。

```
server {
    listen 80;
    server_name example.com;
    index index.php;
    error_log /path/to/example.error.log;
    access_log /path/to/example.access.log;
    root /path/to/public;

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location ~ \.php {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param SCRIPT_NAME $fastcgi_script_name;
        fastcgi_index index.php;
        fastcgi_pass 127.0.0.1:9000;
    }
}
```

HipHop Virtual Machine

Your HipHop Virtual Machine configuration file should contain this code (along with other settings you may need). Be sure you change the `SourceRoot` setting to point to your Slim app's document root directory.

```
Server {
    SourceRoot = /path/to/public/directory
}

ServerVariables {
    SCRIPT_NAME = /index.php
}

VirtualHost {
    * {
        Pattern = .*
        RewriteRules {
                * {
                        pattern = ^(.*)$
                        to = index.php/$1
                        qsa = true
                }
        }
    }
}
```

IIS

务必确保 `Web.config` 和 `index.php` 这两个文件位于同一个可公开访问的目录中。 这个 `Web.config` 文件必须包含以下代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="slim" patternSyntax="Wildcard">
                    <match url="*" />
                    <conditions>
                        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true"
                        <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="t
                    </conditions>
                    <action type="Rewrite" url="index.php" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

lighttpd

Your lighttpd configuration file should contain this code (along with other settings you may need). This code requires lighttpd >= 1.4.24.

```
url.rewrite-if-not-file = ("(.*)" => "/index.php/$0")
```

This assumes that Slim's `index.php` is in the root folder of your project (www root).

教程

First Application Walkthrough

· 本页延后翻译

如果你在寻找创建一款非常简单的 Slim 应用程序的流程，来这里算是找对地方了（在这个应用程序中，没有用到 Twig，但用到了 MonoLog 和一款 PDO 数据库连接器）。你可以在此教程中学习如何构建这个示例应用程序，也可以按自己的需要去改写某个步骤。

在你开始之前：这里有一个 skeleton 项目可以帮助你快速创建一个应用程序样板，你可以使用它直接开始创建你的应用程序，而不用再纠结应用程序的组织结构了。

> 此教程贯穿构建示例应用程序的全流程。如果有需求，可以在 Github 上浏览它的项目源代码。

开始

Start by making a folder for your project (mine is called `project`, because naming things is hard). I like to reserve the top level for things-that-are-not-code and then have a folder for source code, and a folder inside that which is my webroot, so my initial structure looks like this:

```
.
├── project
│   └── src
│       └── public
```

Installing Slim Framework

Composer is the best way to install Slim Framework. If you don't have it already, you can follow the installation instructions, in my project I've just downloaded the `composer.phar` into my `src/` directory and I'll use it locally. So my first command looks like this (I'm in the `src/` directory):

```
php composer.phar require slim/slim
```

This does two things:

- Add the Slim Framework dependency to `composer.json` (in my case it creates the file for me as I don't already have one, it's safe to run this if you do already have a `composer.json` file)
- Run `composer install` so that those dependencies are actually available to use in your

application

If you look inside the project directory now, you'll see that you have a `vendor/` folder with all the library code in it. There are also two new files: `composer.json` and `composer.lock`. This would be a great time to get our source control setup correct as well: when working with composer, we always exclude the `vendor/` directory, but both `composer.json` and `composer.lock` should be included under source control. Since I'm using `composer.phar` in this directory I'm going to include it in my repo as well; you could equally install the `composer` command on all the systems that need it.

To set up the git ignore correctly, create a file called `src/.gitignore` and add the following single line to the file:

```
vendor/*
```

Now git won't prompt you to add the files in `vendor/` to the repository - we don't want to do this because we're letting composer manage these dependencies rather than including them in our source control repository.

Create The Application

There's a really excellent and minimal example of an `index.php` for Slim Framework on the project homepage so we'll use that as our starting point. Put the following code into `src/public/index.php`:

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require '../vendor/autoload.php';

$app = new \Slim\App;
$app->get('/hello/{name}', function (Request $request, Response $response) {
    $name = $request->getAttribute('name');
    $response->getBody()->write("Hello, $name");

    return $response;
});
$app->run();
```

We just pasted a load of code … let's take a look at what it does.

The `use` statements at the top of the script are just bringing the `Request` and `Response` classes into our script so we don't have to refer to them by their long-winded names. Slim framework supports PSR-7 which is the PHP standard for HTTP messaging, so you'll notice as you build your application that the `Request` and `Response` objects are something you see often. This is a modern and excellent approach to writing web applications.

Next we include the `vendor/autoload.php` file - this is created by Composer and allows us to refer to the Slim and other related dependencies we installed earlier. Look out that if you're using the same file structure as me then the `vendor/` directory is one level up from your `index.php` and you may need to adjust the path as I did above.

Finally we create the `$app` object which is the start of the Slim goodness. The `$app->get()` call is our first "route" - when we make a GET request to `/hello/someone` then this is the code that will respond to it. **Don't forget** you need that final `$app->run()` line to tell Slim that we're done configuring and it's time to get on with the main event.

Now we have an application, we'll need to run it. I'll cover two options: the built-in PHP webserver, and an Apache virtual host setup.

Run Your Application With PHP's Webserver

This is my preferred "quick start" option because it doesn't rely on anything else! From the `src/public` directory run the command:

```
php -S localhost:8080
```

This will make your application available at http://localhost:8080 (if you're already using port 8080 on your machine, you'll get a warning. Just pick a different port number, PHP doesn't care what you bind it to).

Note you'll get an error message about "Page Not Found" at this URL - but it's an error message **from** Slim, so this is expected. Try http://localhost:8080/hello/joebloggs instead :)

Run Your Application With Apache

To get this set up on a standard LAMP stack, we'll need a couple of extra ingredients: some virtual host configuration, and one rewrite rule.

The vhost configuration should be fairly straightforward; we don't need anything special here. Copy your existing default vhost configuration and set the `ServerName` to be how you want to refer to your project. For example you can set:

```
ServerName slimproject.dev
```

Then you'll also want to set the `DocumentRoot` to point to the `public/` directory of your project, something like this (edit the existing line):

```
DocumentRoot    /home/lorna/projects/slim/project/src/public/
```

Don't forget to restart apache now you've changed the configuration!

I also have a `.htaccess` file in my `src/public` directory; this relies on Apache's rewrite module being enabled and simply makes all web requests go to index.php so that Slim can then handle all the routing for us. Here's my `.htaccess` file:

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule . index.php [L]
```

With this setup, just remember to use http://slimproject.dev instead of http://localhost:8080 in the other examples in this tutorial. The same health warning as above applies: you'll see an error page at http://slimproject.dev but crucially it's *Slim's* error page. If you go to http://slimproject.dev/hello/joebloggs then something better should happen.

Configuration and Autoloaders

Now we've set up the platform, we can start getting everything we need in place in the application itself.

Add Config Settings to Your Application

The initial example uses all the Slim defaults, but we can easily add configuration to our application when we create it. There are a few options but here I've just created an array of config options and then told Slim to take its settings from here when I create it.

First the configuration itself:

```
$config['displayErrorDetails'] = true;

$config['db']['host']   = "localhost";
$config['db']['user']   = "user";
$config['db']['pass']   = "password";
$config['db']['dbname'] = "exampleapp";
```

The first line is the most important! Turn this on in development mode to get information about errors (without it, Slim will at least log errors so if you're using the built in PHP webserver then you'll see them in the console output which is helpful). The other settings here are not specific keys/values, they're just some data that I want to be able to access later.

Now to feed this into Slim, we need to *change* where we create the `Slim/App` object so that it now looks like this:

```
$app = new \Slim\App(["settings" => $config]);
```

We'll be able to access any settings we put into that `$config` array from our application later on.

Set up Autoloading for Your Own Classes

We already added the composer autoloader file, but what about the code that we write that isn't part of the composer libraries? One option is to use Composer to manage autoloading rules which is a great solution, but you can also add your own autoloader if you want to.

My setup is pretty simple since I only have a few extra classes, they're just in the global namespace, and the files are in the `src/classes/` directory. So to add the autoloader, I have this block of code after the `vendor/autoload.php` file is required:

```
spl_autoload_register(function ($classname) {
    require ("../classes/" . $classname . ".php");
});
```

The spl_autoload_register function accepts the (namespaced) class name and then by the end of the function, the class code should have been included.

Add Dependencies

Most applications will have some dependencies, and Slim handles them nicely using a DIC (Dependency Injection Container) built on Pimple. This example will use both Monolog and a PDO connection to MySQL.

The idea of the dependency injection container is that you configure the container to be able to load the dependencies that your application needs, when it needs them. Once the DIC has created/assembled the dependencies, it stores them and can supply them again later if needed.

To get the container, we can add the following after the line where we create `$app` and before we start to register the routes in our application:

```
$container = $app->getContainer();
```

Now we have the `Slim\Container` object, we can add our services to it.

Use Monolog In Your Application

If you're not already familiar with Monolog, it's an excellent logging framework for PHP applications, which is why I'm going to use it here. First of all, get the Monlog library installed via Composer:

```
php composer.phar require monolog/monolog
```

The dependency is named `logger` and the code to add it looks like this:

```
$container['logger'] = function($c) {
    $logger = new \Monolog\Logger('my_logger');
    $file_handler = new \Monolog\Handler\StreamHandler("../logs/app.log");
    $logger->pushHandler($file_handler);
    return $logger;
};
```

We're adding an element to the container, which is itself an anonymous function (the `$c` that is passed in is the container itself so you can acess other dependencies if you need to). This will be called when we try to access this dependency for the first time; the code here does the setup of the dependency. Next time we try to access the same dependence, the same object that was created the first time will be used the next time.

My Monolog config here is fairly light; just setting up the application to log all errors to a file called `logs/app.log` (remember this path is from the point of view of where the script is running, i.e. `index.php`).

With the logger in place, I can use it from inside my route code with a line like this:

```
 $this->logger->addInfo("Something interesting happened");
```

Having good application logging is a really important foundation for any application so I'd always recommend putting something like this in place. This allows you to add as much or as little debugging as you want, and by using the appropriate log levels with each message, you can have as much or as little detail as is appropriate for what you're doing in any one moment.

Add A Database Connection

There are many database libraries available for PHP, but this example uses PDO - this is available in PHP as standard so it's probably useful in every project, or you can use your own libraries by adapting the examples below.

Exactly as we did for adding Monolog to the DIC, we'll add an anonymous function that sets up the dependency, in this case called `db` :

```
$container['db'] = function ($c) {
    $db = $c['settings']['db'];
    $pdo = new PDO("mysql:host=" . $db['host'] . ";dbname=" . $db['dbname'],
        $db['user'], $db['pass']);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
    return $pdo;
};
```

Remember the config that we added into our app earlier? Well, this is where we use it - the container knows how to access our settings, and so we can grab our configuration very easily from here. With the config, we create the `PDO` object (remember this will throw a `PDOException` if it fails and you might like to handle that here) so that we can connect to the database. I've included two `setAttribute()` calls that really aren't necessary but I find these two settings make PDO itself much more usable as a library so I left the settings in this example so you can use them too! Finally, we return our connection object.

Again, we can access our dependencies with just `$this->` and then the name of the dependency we want which in this case is `$this->db`, so there is code in my application that looks something like:

```
$mapper = new TicketMapper($this->db);
```

This will fetch the `db` dependency from the DIC, creating it if necessary, and in this example just allows me to pass the `PDO` object straight into my mapper class.

Create Routes

"Routes" are the URL patterns that we'll describe and attach functionality to. Slim doesn't use any automatic mapping or URL formulae so you can make any route pattern you like map onto any function you like, it's very flexible. Routes can be linked to a particular HTTP verb (such as GET or POST), or more than one verb.

As a first example, here's the code for making a GET request to `/tickets` which lists the tickets in my bug tracker example application. It just spits out the variables since we haven't added any views to our application yet:

```
$app->get('/tickets', function (Request $request, Response $response) {
    $this->logger->addInfo("Ticket list");
    $mapper = new TicketMapper($this->db);
    $tickets = $mapper->getTickets();

    $response->getBody()->write(var_export($tickets, true));
    return $response;
});
```

The use of `$app->get()` here means that this route is only available for GET requests; there's an equivalent `$app->post()` call that also takes the route pattern and a callback for POST requests. There are also methods for other verbs - and also the `map()` function for situations where more than one verb should use the same code for a particular route.

Slim routes match in the order they are declared, so if you have a route which could overlap another route, you need to put the most specific one first. Slim will throw an exception if there's a problem, for example in this application I have both `/ticket/new` and `/ticket/{id}` and they need to be declared in that order otherwise the routing will think that "new" is an ID!

In this example application, all the routes are in `index.php` but in practice this can make for a rather long and unwieldy file! It's fine to refactor your application to put routes into a different file or files, or just register a set of routes with callbacks that are actually declared elsewhere.

All route callbacks accept three parameters (the third one is optional):

- Request: this contains all the information about the incoming request, headers, variables, etc.
- Response: we can add output and headers to this and, once complete, it will be turned into the HTTP response that the client receives
- Arguments: the named placeholders from the URL (more on those in just a moment), this is optional and is usually omitted if there aren't any

This emphasis on Request and Response illustrates Slim 3 being based on the PSR-7 standard for HTTP Messaging. Using the Request and Response object also makes the application more testable as we don't need to make **actual** requests and responses, we can just set up the objects as desired.

Routes with Named Placeholders

Sometimes, our URLs have variables in them that we want to use in our application. In my bug tracking example, I want to have URLs like `/ticket/42` to refer to the ticket - and Slim has an easy way of parsing out the "42" bit and making it available for easy use in the code. Here's the route that does exactly that:

```
$app->get('/ticket/{id}', function (Request $request, Response $response, $args) {
    $ticket_id = (int)$args['id'];
    $mapper = new TicketMapper($this->db);
    $ticket = $mapper->getTicketById($ticket_id);

    $response->getBody()->write(var_export($ticket, true));
    return $response;
});
```

Look at where the route itself is defined: we write it as `/ticket/{id}`. When we do this, the route will take the portion of the URL from where the `{id}` is declared, and it becomes available as `$args['id']` inside the callback.

Using GET Parameters

Since GET and POST send data in such different ways, then the way that we get that data from the Request object differs hugely in Slim.

It is possible to get all the query parameters from a request by doing `$request->getQueryParams()` which will return an associative array. So for the URL `/tickets?sort=date&order=desc` we'd get an associative array like:

```
["sort" => "date", "order" => "desc"]
```

These can then be used (after validating of course) inside your callback.

Working with POST Data

When working with incoming data, we can find this in the body. We've already seen how we can parse data from the URL and how to obtain the GET variables by doing `$request->getQueryParams()` but what about POST data? The POST request data can be found in the body of the request, and Slim has some good built in helpers to make it easier to get the information in a useful format.

For data that comes from a web form, Slim will turn that into an array. My tickets example application has a form for creating new tickets that just sends two fields: "title" and "description". Here is the first part of the route that receives that data, note that for a POST route use `$app->post()` rather than `$app->get()`:

```
$app->post('/ticket/new', function (Request $request, Response $response) {
    $data = $request->getParsedBody();
    $ticket_data = [];
    $ticket_data['title'] = filter_var($data['title'], FILTER_SANITIZE_STRING);
    $ticket_data['description'] = filter_var($data['description'], FILTER_SANITIZE_STRING
    // ...
```

The call to `$request->getParsedBody()` asks Slim to look at the request and the `Content-Type` headers of that request, then do something smart and useful with the body. In this example it's just a form post and so the resulting `$data` array looks very similar to what we'd expect from `$_POST` - and we can go ahead and use the filter extension to check the

value is aceptable before we use it. A huge advantage of using the built in Slim methods is that we can test things by injecting different request objects - if we were to use `$_POST` directly, we aren’t able to do that.

What’s really neat here is that if you’re building an API or writing AJAX endpoints, for example, it’s super easy to work with data formats that arrive by POST but which aren’t a web form. As long as the `Content-Type` header is set correctly, Slim will parse a JSON payload into an array and you can access it exactly the same way: by using `$request->getParsedBody()`.

Views and Templates

Slim doesn’t have an opinion on the views that you should use, although there are some options that are ready to plug in. Your best choices are either Twig or plain old PHP. Both options have pros and cons: if you’re already familiar with Twig then it offers lots of excellent features and functionality such as layouts - but if you’re not already using Twig, it can be a large learning curve overhead to add to a microframework project. If you’re looking for something dirt simple then the PHP views might be for you! I picked PHP for this example project, but if you’re familiar with Twig then feel free to use that; the basics are mostly the same.

Since we’ll be using the PHP views, we’ll need to add this dependency to our project via Composer. The command looks like this (similar to the ones you’ve already seen):

```
php composer.phar require slim/php-view
```

In order to be able to render the view, we’ll first need to create a view and make it available to our application; we do that by adding it to the DIC. The code we need goes with the other DIC additions near the top of `src/public/index.php` and it looks like this:

```
$container['view'] = new \Slim\Views\PhpRenderer("../templates/");
```

Now we have a `view` element in the DIC, and by default it will look for its templates in the `src/templates/` directory. We can use it to render templates in our actions - here’s the ticket list route again, this time including the call to pass data into the template and render it:

```
$app->get('/tickets', function (Request $request, Response $response) {
    $this->logger->addInfo("Ticket list");
    $mapper = new TicketMapper($this->db);
    $tickets = $mapper->getTickets();

    $response = $this->view->render($response, "tickets.phtml", ["tickets" => $tickets]);
    return $response;
});
```

The only new part here is the penultimate line where we set the `$response` variable. Now that the `view` is in the DIC, we can refer to it as `$this->view`. Calling `render()` needs us to supply three arguments: the `$response` to use, the template file (inside the default templates directory), and any data we want to pass in. Response objects are *immutable* which means that the call to `render()` won't update the response object; instead it will return us a new object which is why it needs to be captured like this. This is always true when you operate on the response object.

When passing the data to templates, you can add as many elements to the array as you want to make available in the template. The keys of the array are the variables that the data will exist in once we get to the template itself.

As an example, here's a snippet from the template that displays the ticket list (i.e. the code from `src/templates/tickets.phtml` - which uses Pure.css to help cover my lack of frontend skills):

```
<h1>All Tickets</h1>

<p><a href="/ticket/new">Add new ticket</a></p>

<table class="pure-table">
    <tr>
        <th>Title</th>
        <th>Component</th>
        <th>Description</th>
        <th>Actions</th>
    </tr>

<?php foreach($data['tickets'] as $ticket): ?>

    <tr>
        <td><?=$ticket->getTitle() ?></td>
        <td><?=$ticket->getComponent() ?></td>
        <td><?=$ticket->getShortDescription() ?> ...</td>
        <td>
            <a href="<?=$router->pathFor('ticket-detail', ['id' => $ticket->getId()])?>">
        </td>
    </tr>

<?php endforeach; ?>
</table>
```

In this case, `$tickets` is actually a `TicketEntity` class with getters and setters, but if you passed in an array, you'd be able to access it using array rather than object notation here.

Did you notice something fun going on with `$router->pathFor()` right at the end of the example? Let's talk about named routes next :)

Easy URL Building with Named Routes

When we create a route, we can give it a name by calling `->setName()` on the route object. In this case, I am adding the name to the route that lets me view an individual ticket so that I can quickly create the right URL for a ticket by just giving the name of the route, so my code now looks something like this (just the changed bits shown here):

```
$app->get('/ticket/{id}', function (Request $request, Response $response, $args) {
    // ...
})->setName("ticket-detail");
```

To use this in my template, I need to make the router available in the template that's going to want to create this URL, so I've amended the `tickets/` route to pass a router through to the template by changing the render line to look like this:

```
 $response = $this->view->render($response, "tickets.phtml", ["tickets" => $tickets, "rou
```

With the `/tickets/{id}` route having a friendly name, and the router now available in our template, this is is what makes the `pathFor()` call in our template work. By supplying the `id`, this gets used as a named placeholder in the URL pattern, and the correct URL for linking to that route with those values is created. This feature is brilliant for readable template URLs and is even better if you ever need to change a URL format for any reason - no need to grep templates to see where it's used. This approach is definitely recomended, especially for links you'll use a lot.

Where Next?

This article gave a walkthrough of how to get set up with a simple application of your own, which I hope will let you get quickly started, see some working examples, and build something awesome.

From here, I'd recommend you take a look at the other parts of the project documentation for anything you need that wasn't already covered or that you want to see an alternative example of. A great next step would be to take a look at the Middleware section - this technique is how we layer up our application and add functionality such as authentication which can be applied to multiple routes.

概念

PSR 7 与值对象（Value Objects）

Slim 为其请求和响应对象支持了 PSR-7 接口。这使得 Slim 更加灵活了，因为它可以使用 任意 PSR-7 实现方法。例如，一个 Slim 应用程序不必返回一个 `\Slim\Http\Response` 的实例。它可以这样子，举例来说，比如返回一个 `\GuzzleHttp\Psr7\CachingStream` 的实例，或者，由 `\GuzzleHttp\Psr7\stream_for()` 函数返回的任意实例。

Slim 提供了它自有的 PSR-7 实现方法，使其可以开箱即用。然而，你可以自由地用第三方实现方法来替换 Slim 的默认 PSR 7 对象。只需覆写应用容器的 `request` 和 `response` 服务，这样它们就能分别返回一个 `\Psr\Http\Message\ServerRequestInterface` 和 `\Psr\Http\Message\ResponseInterface` 的实例。

值对象（Value objects）

Slim 的请求和响应对象是不可改变的值对象。. 只能通过请求一个有属性值更新的克隆版本来改变它们。值对象有一个额定的开销，因为它们必须在更新时进行克隆。这个开销并不会以任何有实际意义的方式影响到性能。

你可以通过调用任意 PSR 7 接口方法来请求值对象的拷贝（这些方法通常带有 `with` 前缀）。例如，一个 PSR 7 响应对象有一个 `withHeader($name, $value)` 方法，它返回一个克隆的带有新 HTTP 头的值对象。

```
<?php
$app = new \Slim\App;
$app->get('/foo', function ($req, $res, $args) {
    return $res->withHeader(
        'Content-Type',
        'application/json'
    );
});
$app->run();
```

PSR 7 接口提供了以下方法来转换请求和响应对象:

- `withProtocolVersion($version)`
- `withHeader($name, $value)`
- `withAddedHeader($name, $value)`
- `withoutHeader($name)`
- `withBody(StreamInterface $body)`

PSR 7 接口提供了以下方法来转换请求对象：

- `withMethod($method)`
- `withUri(UriInterface $uri, $preserveHost = false)`

- `withCookieParams(array $cookies)`
- `withQueryParams(array $query)`
- `withUploadedFiles(array $uploadedFiles)`
- `withParsedBody($data)`
- `withAttribute($name, $value)`
- `withoutAttribute($name)`

PSR 7 接口提供了以下方法来转换响应对象：

- `withStatus($code, $reasonPhrase = '')`

访问 PSR-7 文档 了解关于这些方法的更多信息吧。

中间件

你可以在你的 Slim 应用之前（*before*） 和 之后（*after*） 运行代码来处理你认为合适的请求和响应对象。这就叫做中间件。为什么要这么做呢？比如你想保护你的应用不遭受跨站请求伪造。也许你想在应用程序运行前验证请求。中间件对这些场景的处理简直完美。

什么是中间件？

从技术上来讲，中间件是一个接收三个参数的可回调（*callable*）对象：

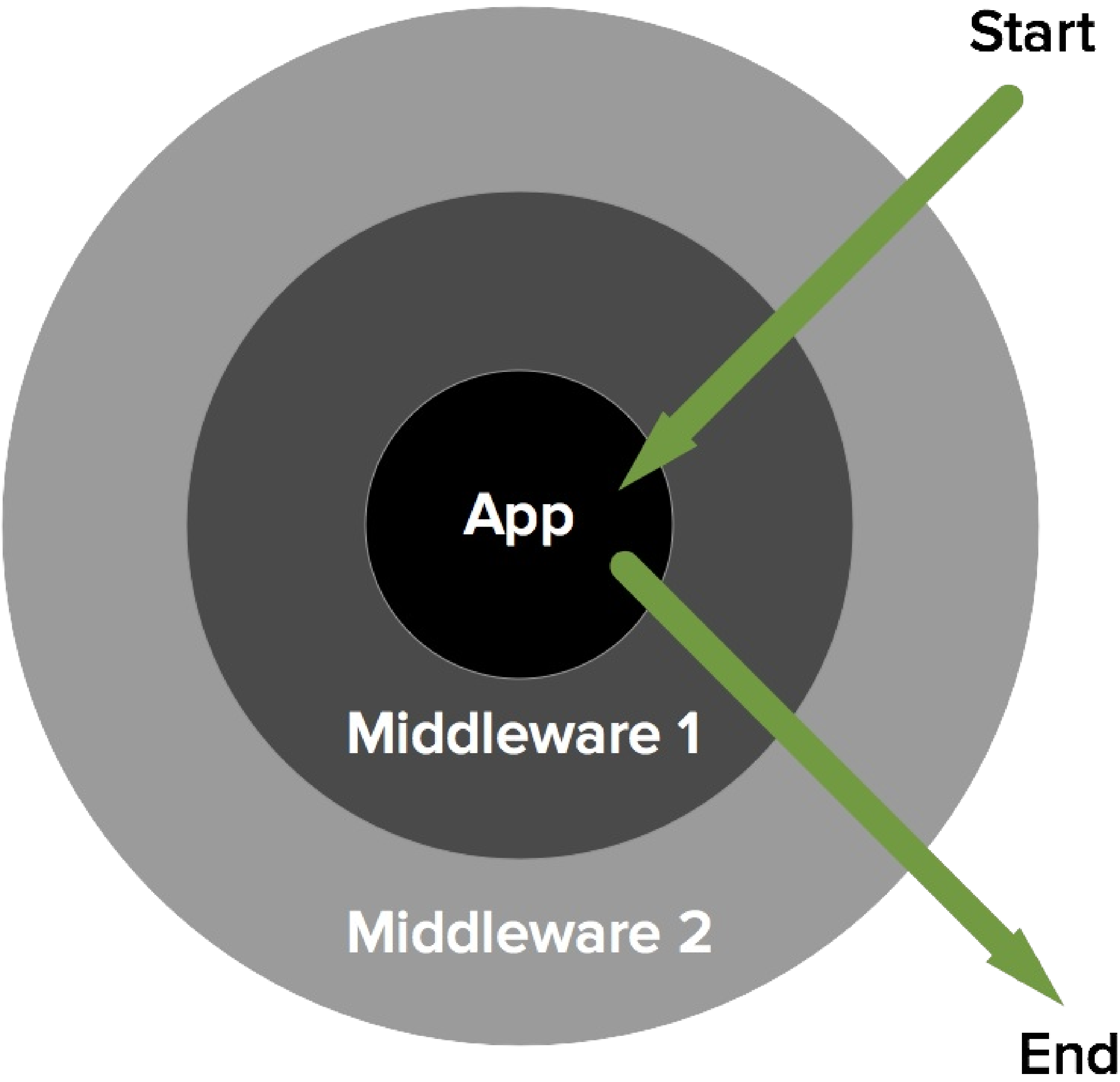
1. `\Psr\Http\Message\ServerRequestInterface` - PSR7 请求对象
2. `\Psr\Http\Message\ResponseInterface` - PSR7 响应对象
3. `callable` - 下一层中间件的回调对象

它可以做任何与这些对象相应的事情。唯一硬性要求就是中间件必须返回一个 `\Psr\Http\Message\ResponseInterface` 的实例。 每个中间件都 应当调用下一层中间件，并讲请求和响应对象作为参数传递给它（下一层中间件）。

中间件是如何工作的？

不同的框架使用中间件的方式不同。在 Slim 框架中，中间件层以同心圆的方式包裹着核心应用。由于新增的中间件层总会包裹所有已经存在的中间件层。当添加更多的中间件层时同心圆结构会不断的向外扩展。

当 Slim 应用运行时，请求对象和响应对象从外到内穿过中间件结构。它们首先进入最外层的中间件，然后然后进入下一层，（以此类推）。直到最后它们到达了 Slim 应用程序自身。在 Slim 应用分派了对应的路由后，作为结果的响应对象离开 Slim 应用，然后从内到外穿过中间件结构。最终，最后出来的响应对象离开了最外层的中间件，被序列化为一个原始的 HTTP 响应消息，并返回给 HTTP 客户端。下图清楚的说明了中间件的工作流程：



如何编写中间件？

中间件是一个接收三个参数的可回调（callable）对象，三个参数是：请求对象、响应对象以及下一层中间件。中间件必须返回一个 `\Psr\Http\Message\ResponseInterface` 的实例。

闭包中间件示例。

这个示例中间件是一个闭包。

```
<?php
/**
 * Example middleware closure
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $request  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $response PSR7 response
 * @param  callable                                 $next     Next middleware
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
function ($request, $response, $next) {
    $response->getBody()->write('BEFORE');
    $response = $next($request, $response);
    $response->getBody()->write('AFTER');

    return $response;
};
```

可调用类的中间件示例。

这个示例中间件是一个实现了 `__invoke()` 魔术方法的可调用类。

```
<?php
class ExampleMiddleware
{
    /**
 * Example middleware invokable class
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $request  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $response PSR7 response
 * @param  callable                                 $next     Next middleware
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
    public function __invoke($request, $response, $next)
    {
        $response->getBody()->write('BEFORE');
        $response = $next($request, $response);
        $response->getBody()->write('AFTER');

        return $response;
    }
}
```

要使用这个类作为中间件，你可以使用 `->add( new ExampleMiddleware() );` 函数连接在 `$app` , `Route` , 或 `group()` 之后，在下面的代码中，它们中的任意一个都可以表示 $subject。

```
$subject->add( new ExampleMiddleware() );
```

如何添加中间件？

你可以在 Slim 应用中添加中间件，或者添加到一个单独的 Slim 应用路由，或者是路由组。所有场景都能接受相同的中间件和实现相同的中间件接口。

应用程序中间件

应用程序中间件被每个传入（*incoming*） HTTP 请求调用。应用中间件可以通过 Slim 应用实例的 `add()` 方法来添加。下面这是一个添加上面那个闭包中间件的例子：

```
<?php
$app = new \Slim\App();

$app->add(function ($request, $response, $next) {
    $response->getBody()->write('BEFORE');
    $response = $next($request, $response);
    $response->getBody()->write('AFTER');

    return $response;
});

$app->get('/', function ($req, $res, $args) {
    echo ' Hello ';
});

$app->run();
```

输出的 HTTP 响应主体为：

```
BEFORE Hello AFTER
```

路由中间件

路由中间件只有在当前 HTTP 请求的方法和 URI 都与中间件所在路由相匹配时才会被调用。路由中间件会在你调用了任意 Slim 应用的路由方法（e.g., `get()` 或 `post()` ）后被立即指定。 每个路由方法返回一个 `\Slim\Route` 的实例，这个类提供了相同的中间件接口作为 Slim 应用的实例。使用路由实例的 `add()` 方法将中间件添加到路由中。下面这是一个添加上面那个闭包中间件的例子：

```
<?php
$app = new \Slim\App();

$mw = function ($request, $response, $next) {
    $response->getBody()->write('BEFORE');
    $response = $next($request, $response);
    $response->getBody()->write('AFTER');

    return $response;
};

$app->get('/', function ($req, $res, $args) {
    echo ' Hello ';
})->add($mw);

$app->run();
```

输出的 HTTP 响应主体为：

```
BEFORE Hello AFTER
```

Group Middleware

除了整个应用，以及标准的路由可用接收中间件，还有 `group()` 多路由定义功能同样允许在其内部存在独立的路由。路由组中间件只有在其路由与已定义的 HTTP 请求和 URI 中的一个相匹配时才会被调用。要在回调中添加中间件，以及整组中间件，通过在 `group()` 方法后面连接 `add()` 来设置。

下面的示例程序，在一组 URL 处理程序（url-handlers）中使用了回调中间件。

```
<?php

require_once __DIR__.'/vendor/autoload.php';

$app = new \Slim\App();

$app->get('/', function ($request, $response) {
    return $response->getBody()->write('Hello World');
});

$app->group('/utils', function () use ($app) {
    $app->get('/date', function ($request, $response) {
        return $response->getBody()->write(date('Y-m-d H:i:s'));
    });
    $app->get('/time', function ($request, $response) {
        return $response->getBody()->write(time());
    });
})->add(function ($request, $response, $next) {
    $response->getBody()->write('It is now ');
    $response = $next($request, $response);
    $response->getBody()->write('. Enjoy!');

    return $response;
});
```

在调用 `/utils/date` 方法时，将输出类似下面这样子的字符串：

```
It is now 2015-07-06 03:11:01\. Enjoy!
```

访问 `/utils/time` 将会输出类似下面这样子的字符串：

```
It is now 1436148762\. Enjoy!
```

但访问 `/` 域名根目录*(domain-root)*时，将会有如下输出，因为并没有分配中间件。

```
Hello World
```

依赖容器（Dependency Container）

Slim 使用依赖容器来准备、管理和注入应用程序的相关依赖。Slim 支持 Container-Interop 接口实现的容器。你可以使用 Slim 的内置容器（基于 Pimple）或者第三方的容器，比如 Acclimate 或 PHP-DI。

如何使用容器

你并不必须提供一个依赖容器。如果你提供了，那么，你必须注入此容器的实例到 Slim 应用程序的构造函数中。

```
$container = new \Slim\Container;
$app = new \Slim\App($container);
```

你可以显式或隐式地从依赖容器中获取服务。你可以像下面这样子从 Slim 应用程序的路由中获取一个显示的容器实例。

```
/**
 * Example GET route
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $req  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $res  PSR7 response
 * @param  array                                    $args Route parameters
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
$app->get('/foo', function ($req, $res, $args) {
 $myService = $this->get('myService');

 return $res;
});
```

你可以这样隐式地从容器中取得服务：

```
/**
 * Example GET route
 *
 * @param  \Psr\Http\Message\ServerRequestInterface $req  PSR7 request
 * @param  \Psr\Http\Message\ResponseInterface      $res  PSR7 response
 * @param  array                                    $args Route parameters
 *
 * @return \Psr\Http\Message\ResponseInterface
 */
$app->get('/foo', function ($req, $res, $args) {
 $myService = $this->myService;

 return $res;
});
```

Slim uses `__get()` and `__isset()` magic methods that defer to the application's container for all properties that do not already exist on the application instance.

必需的服务

你的容器必须实现这些必需的服务。如果你使用的是 Slim 内置的容器，这些服务都是已经准备好了的。如果你选择使用第三方容器，那么你必须自己来实现这些服务。

settings

应用程序设置项的关联数组（Associative array），包括以下关键字：

- `httpVersion`
- `responseChunkSize`
- `outputBuffering`
- `determineRouteBeforeAppMiddleware`.
- `displayErrorDetails`.

environment

`\Slim\Interfaces\Http\EnvironmentInterface` 的实例.

request

`\Psr\Http\Message\ServerRequestInterface` 的实例.

response

`\Psr\Http\Message\ResponseInterface` 的实例.

router

`\Slim\Interfaces\RouterInterface` 的实例.

foundHandler

`\Slim\Interfaces\InvocationStrategyInterface` 的实例.

phpErrorHandler

PHP 7 错误被抛出时调用的 Callable。这个 callable 必须返回一个 `\Psr\Http\Message\ResponseInterface` 的实例，并接收三个参数：

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`
3. `\Error`

errorHandler

抛出异常时调用的 Callable。这个 callable 必须返回一个 `\Psr\Http\Message\ResponseInterface` 的实例，并接收三个参数：

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`
3. `\Exception`

notFoundHandler

如果当前的 HTTP 请求 URI 未能匹配到应用程序路由，则调用这个 Callable。这个 callable 必须返回一个 `\Psr\Http\Message\ResponseInterface` 的实例，并接收三个参数：

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`

notAllowedHandler

如果一个应用程序路由匹配到当前 HTTP 请求的路径而不是它的方法，则调用这个 Callable。这个 callable 必须 返回一个 `\Psr\Http\Message\ResponseInterface` 的实例并接收三个参数：

1. `\Psr\Http\Message\ServerRequestInterface`
2. `\Psr\Http\Message\ResponseInterface`
3. Array of allowed HTTP methods

callableResolver

`\Slim\Interfaces\CallableResolverInterface` 的实例.

Application

Application ，（或者 `Slim\App` ） 是你的 Slim 应用程序的入口，它被用于注册那些链接到回调和控制器的路由。

```
// 实例化 App 对象
$app = new \Slim\App();

// 添加路由回调
$app->get('/', function ($request, $response, $args) {
    return $response->withStatus(200)->write('Hello World!');
});

// 运行应用
$app->run();
```

Application 配置

Application 只接收一个参数。该参数可以是容器实例或者用于配置自动创建的默认容器的数组。

Slim 还用到了一系列的设置项。它们被存放在 `settings` 配置关键字中。你还可以添加你的应用程序私有的设置项。

例如，我们可以将 Slim的设置项 `displayErrorDetails` 设置为 true，并配置 Monolog，像这样：

```
$config = [
    'settings' => [
        'displayErrorDetails' => true,

        'logger' => [
            'name' => 'slim-app',
            'level' => Monolog\Logger::DEBUG,
            'path' => __DIR__ . '/../logs/app.log',
        ],
    ],
];
$app = new \Slim\App($config);
```

获取 Settings

由于设置项都被存放在依赖注入容器中，所以你可以通过容器工厂方法（container factories）的 `settings` 关键字来访问它们。例如：

```
$settings = $container->get('settings')['logger'];
```

还可以在路由回调（route callable）中通过 `$this` 来访问它们：

```
$app->get('/', function ($request, $response, $args) {
    $loggerSettings = $this->get('settings')['logger'];
    // ...
});
```

Slim 的默认设置

Slim 拥有以下默认设置，你可以按需覆写它们：

`httpVersion`

HTTP 响应对象使用的 HTTP 协议版本 (默认: `'1.1'`)

`responseChunkSize`

从响应体读取并发送到浏览器的数据包的大小。(默认: `4096`)

`outputBuffering`

当设置为 `false` 时，那么没有输出缓冲被启用。如果 `'append'` 或 `'prepend'` ，那么任意 `echo` 或 `print` 状态都会被捕获，并且会 appended 或 prepended 到从路由回调（route callable）中返回的响应中。(默认: `'append'`)

`determineRouteBeforeAppMiddleware`

当设置为 true 时，那么在中间件执行前即已确定路由是否正确。这意味着你如果有需要，可以在中间件中检查路由参数。(默认: `false`)

`displayErrorDetails`

当设置为 true 时, 关于异常的附加信息都会通过 默认的错误处理器 显示出来。(默认: `false`)

HTTP 请求

Slim 应用程序的路由和中间件给出了一个 PSR 7 请求对象，它表示当前的 HTTP 请求是由 Web 服务器 接收到的。该请求对象遵循 PSR 7 服务器请求接口（ServerRequestInterface ）实现，因此你可以检查和操作该 HTTP 请求对象的方法、头和体。

如何获取请求对象

该 PSR 7 请求对象作为路由回调的第一个参数注入到你的 Slim 应用程序的路由中，像这样：

```
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->get('/foo', function (ServerRequestInterface $request, ResponseInterface $response)
    // 使用 PSR 7 $request 对象

    return $response;
});
$app->run();
```

Figure 1: 将 PSR 7 请求注入到应用程序的路由回调中。

该 PSR 7 请求对象作为中间件 callable 的第一个参数注入到 Slim 应用程序的中间件中，像这样：

```
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->add(function (ServerRequestInterface $request, ResponseInterface $response, callabl
    // Use the PSR 7 $request object

    return $next($request, $response);
});
// Define app routes...
$app->run();
```

Figure 2: 注入 PSR 7 请求到应用程序中间件

请求的方法

每个 HTTP 请求都有相应的方法，通常是这些中的一个：

- GET
- POST
- PUT
- DELETE
- HEAD
- PATCH
- OPTIONS

你可以恰当地使用 `getMethod()` 请求对象方法来检查 HTTP 请求方法。

```
$method = $request->getMethod();
```

由于这是一个常见的功能，Slim 的内置 PSR 7 实现方法也提供了这些专有方法，它们返回 `true` 或 `false` 。

- `$request->isGet()`
- `$request->isPost()`
- `$request->isPut()`
- `$request->isDelete()`
- `$request->isHead()`
- `$request->isPatch()`
- `$request->isOptions()`

还可以伪造或覆写这些 HTTP 请求方法。这非常有用，例如你需要在一个只支持 `GET` 或 `POST` 请求的传统浏览器中模拟一个 `PUT` 请求。

有两种方法来覆写 HTTP 请求方法。你可以在一个 `POST` 请求体中引入（include）一个 `_METHOD` 参数。该 HTTP 请求必须使用 `application/x-www-form-urlencoded` 内容类型（content type）。

```
POST /path HTTP/1.1
Host: example.com
Content-type: application/x-www-form-urlencoded
Content-length: 22

data=value&_METHOD=PUT
```

Figure 3: 使用 _METHOD 参数覆写 HTTP 请求。

你也可以使用自定义的 `X-Http-Method-Override` HTTP请求头来覆写 HTTP 请求方法。这个方式可以用于任意 HTTP 请求内容类型（content type）。

```
POST /path HTTP/1.1
Host: example.com
Content-type: application/json
Content-length: 16
X-Http-Method-Override: PUT

{"data":"value"}
```

Figure 4: 使用 X-Http-Method-Override 请求头覆写 HTTP 方法。

你可以使用 PSR 7 请求对象的方法 `getOriginalMethod()` 来提取原有（不是覆写后的）的 HTTP 方法。

请求 URI

每个 HTTP 请求都有一个识别被请求的应用程序资源的 URI 。HTTP 请求 URI 分为这几部分：

- Scheme (e.g. `http` or `https`)
- Host (e.g. `example.com`)
- Port (e.g. `80` or `443`)
- Path (e.g. `/users/1`)
- Query string (e.g. `sort=created&dir=asc`)

你可以使用 `getUri()` 方法来提取 PSR 7 请求对象的 URI ：

```
$uri = $request->getUri();
```

PSR 7 请求对象的 URI 本身就是一个对象，提供了以下方法来检查 HTTP 请求的 URL ：

- `getScheme()`
- `getAuthority()`
- `getUserInfo()`
- `getHost()`
- `getPort()`
- `getPath()`
- `getBasePath()`
- `getQuery()` <small>(返回整个查询字符串，e.g. `a=1&b=2`)</small>
- `getFragment()`
- `getBaseUrl()`

基准路径如果你的 Slim 应用程序的前端控制器放置在文件根目录的物理子目录中，你可以使用 URI 对象的 `getBasePath()` 方法来提取 HTTP 请求的物理基准路径（相对于文件根目录）。如果 Slim 应用程序安装在文件根目录的最上层目录中，它将返回一个空字符串。

请求头

每个 HTTP 请求都有请求头。这些元数据描述了 HTTP 请求，但在请求体中不可见。Slim 的 PSR 7 请求对象提供了几个检查请求头的方法。

获取所有请求头

使用 PSR 7 请求对象的 `getHeaders()` 方法提取所有 HTTP 请求头并放入一个关联数组中。此关联数组的键值是请求头的名称，其值是各请求头对应的由字符串值组成的数值数组。

```
$headers = $request->getHeaders();
foreach ($headers as $name => $values) {
 echo $name . ": " . implode(", ", $values);
}
```

Figure 5: 提取并迭代所有 HTTP 请求头作为一个关联数组。

获取单个请求头

你可以使用 PSR 7 请求对象的 `getHeader($name)` 方法获取一个单独的请求头的值。它将返回一个由指定请求头名称对应的值组成的数组。记住，单独的请求头可不一定只有一个值。

```
$headerValueArray = $request->getHeader('Accept');
```

Figure 6: 获取指定 HTTP 请求的值。

你同样可以使用 PSR 7 请求对象的 `getHeaderLine($name)` 方法提取指定请求头的值，并以逗号分隔。这不同于 `getHeader($name)` 方法，这个方法返回一个由逗号分隔的字符串。

```
$headerValueString = $request->getHeaderLine('Accept');
```

Figure 7: 获取单个请求头的值，返回逗号分隔的字符串。

检测请求头

使用 PSR 7 请求对象的 `hasHeader($name)` 方法检查某请求头是否存在。

```
if ($request->hasHeader('Accept')) {
 // Do something
}
```

Figure 8: Detect presence of a specific HTTP request header.

请求体

每个 HTTP 请求都有一个请求体。如果你的 Slim 应用程序是通过 JSON 或 XML 数据进行通信的，你可以使用 PSR 7 请求对象的 `getParsedBody()` 方法将 HTTP 请求体解析成原生 PHP 格式。Slim 可以解析 JSON, XML, 和 URL-encoded 数据，开箱即用。

```
$parsedBody = $request->getParsedBody();
```

Figure 9: Parse HTTP request body into native PHP format

- JSON 请求通过 `json_decode($input)` 转换成 PHP 对象。
- XML 请求通过 `simplexml_load_string($input)` 转换成 `SimpleXMLElement` 。
- URL-encoded 请求通过 `parse_str($input)` 转换成 PHP 数组。

从技术上说，Slim 的 PSR 7 请求对象将 HTTP 请求体表示为 `\Psr\Http\Message\StreamInterface` 的一个实例。你可以通过使用 PSR 7 请求对象的 `getBody()` 方法获取 HTTP 请求体的 `StreamInterface` 实例。该 `getBody()` 方法在处理未知大小或对于可用内存来说太大的 HTTP 请求时，是个很好的方法。

```
$body = $request->getBody();
```

Figure 10: 获取 HTTP 请求体

生成的 `\Psr\Http\Message\StreamInterface` 实例提供了以下方法来读取或迭代未知的 `资源（resource）` 。

- `getSize()`
- `tell()`
- `eof()`
- `isSeekable()`
- `seek()`
- `rewind()`
- `isWritable()`
- `write($string)`
- `isReadable()`
- `read($length)`
- `getContents()`
- `getMetadata($key = null)`

请求助手/Request Helpers

Slim 的 PSR 7 请求实现方法提供了额外的专有方法来帮助你进一步检查 HTTP 请求。

检测 XHR 请求

你可以使用请求对象的 `isXhr()` 方法来检测 XHR 请求。该方法检测 `X-Requested-With` 请求头，并确保它的值是 `XMLHttpRequest` 。

```
POST /path HTTP/1.1
Host: example.com
Content-type: application/x-www-form-urlencoded
Content-length: 7
X-Requested-With: XMLHttpRequest

foo=bar
```

Figure 11: XHR 请求示例.

```
if ($request->isXhr()) {
 // Do something
}
```

内容类型/Content Type

你可以使用请求对象的 `getContentType()` 方法提取 HTTP 请求的内容类型。它将通过 HTTP 客户端返回 `Content-Type` 头的完整值。

```
$contentType = $request->getContentType();
```

媒体类型/Media Type

你或许不会想要完整的 `Content-Type` 头。加入说你只想要媒体类型？你可以使用响应对象的 `getMediaType()` 方法取得 HTTP 请求的媒体类型。

```
$mediaType = $request->getMediaType();
```

可以使用请求对象的 `getMediaTypeParams()` 方法，以关联数组的形式获取附加的没提类型参数。

```
$mediaParams = $request->getMediaTypeParams();
```

字符集/Character Set

HTTP 请求字符集是最常用的媒体类型参数之一。请求对象提供了一个专用的方法来获取这个媒体类型参数。

```
$charset = $request->getContentCharset();
```

内容长度/Content Length

使用请求对象的 `getContentLength()` 方法获取 HTTP 请求的内容长度。

```
$length = $request->getContentLength();
```

路由对象/Route Object

有时在中间件中你需要路由的参数。

在这个例子中，我们首先检查用户是否登录，然后检查用户有否有权限浏览他们正试图浏览的视频。

```
$app->get('/course/{id}', Video::class.":watch")->add(Permission::class)->add(Auth::class

 //.. In the Permission Class's Invoke
 /** @var $route \Slim\Route */
 $route = $request->getAttribute('route');
 $courseId = $route->getArgument('id');
```

媒体类型解析

如果 HTTP 请求的媒体类型被识别，将通过 `$request->getParsedBody()` 解析成为结构化的可用数据。通常是解析成一个数组，或者是 XML 媒体类型的对象。

以下媒体类型是可识别和解析的：

- application/x-www-form-urlencoded’
- application/json
- application/xml & text/xml

如果你想要用 Slim 来解析其它媒体类型，那么你可以自行解析原生的 HTTP 请求体，或者新注册一个媒体解析器。媒体解析器都是简单的 callable ，它接收一个 `$input` 字符串并返回一个解析后的对象或数组。

在应用程序或者路由中间件中注册新的媒体解析器。注意，你必须在首次尝试访问解析后的 HTTP 请求体前注册该解析器。

例如，要自动解析带有 `text/javascript` 内容类型的 JSON，你可以像这样在中间件中注册媒体解析器：

```
// 添加中间件
$app->add(function ($request, $response, $next) {
    // add media parser
    $request->registerMediaTypeParser(
        "text/javascript",
        function ($input) {
            return json_decode($input, true);
        }
    );

    return $next($request, $response);
});
```

HTTP 响应

Slim 应用程序的路由和中间件给出了一个 PSR 7 响应对象，它表示当前的 HTTP 响应将被返回给客户端。该响应对象遵循 PSR 7 响应接口实现，因此你可以检查和操作该 HTTP 响应的状态、响应头和响应体。

如何获取 HTTP 响应对象

PSR 7 响应对象作为路由回调的第二个参数注入到 Slim 应用程序的路由中：

```
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->get('/foo', function (ServerRequestInterface $request, ResponseInterface $response)
    // Use the PSR 7 $response object
    return $response;
});
$app->run();
```

Figure 1: Inject PSR 7 response into application route callback.

PSR 7 请求对象作为中间件 callable 的第二个参数注入到 Slim 应用程序的中间件：

```
<?php
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\ResponseInterface;

$app = new \Slim\App;
$app->add(function (ServerRequestInterface $request, ResponseInterface $response, callabl
    // Use the PSR 7 $response object
    return $next($request, $response);
});
// Define app routes... $app->run();
```

Figure 2: Inject PSR 7 response into application middleware.

HTTP 响应状态

每个 HTTP 响应都有一个数字 状态编码。状态编码用于识别返回客户端的 HTTP 响应的类型。PSR 7 响应对象的默认状态编码是 `200` (OK)。你可以像这样使用 `getStatusCode()` 方法获取 PSR 7 响应对象的状态编码：

```
$status = $response->getStatusCode();
```

Figure 3: Get response status code.

你可以拷贝一个 PSR 7 响应对象，并指定一个新的状态编码，像这样：

```
$newResponse = $response->withStatus(302);
```

Figure 4: Create response with new status code.

HTTP 响应头

每个 HTTP 响应都有其相应的响应头。这些元数据描述了 HTTP 响应，但在响应体中不可见。Slim 的 PSR 7 响应对象提供了几种检查和操作响应头的方法。

获取所有响应头

使用 PSR 7 响应对象的 `getHeaders()` 方法提取所有 HTTP 响应头并存入一个关联数组中。该关联数组的键名即为响应头的名称，键值是与其响应头对应的值的字符串数组。

```
$headers = $response->getHeaders();
foreach ($headers as $name => $values) {
    echo $name . ": " . implode(", ", $values);
}
```

Figure 5: Fetch and iterate all HTTP response headers as an associative array.

获取单个响应头

使用 PSR 7 响应对象的 `getHeader($name)` 方法获取单个响应头的值。它将返回指定响应头的值组成的数组。记住，单个 HTTP 响应不止一个值。

```
$headerValueArray = $response->getHeader('Vary');
```

Figure 6: Get values for a specific HTTP header.

同样，可以是 PSR 7 响应对象的 `getHeaderLine($name)` 方法获取指定响应头的所有值，由逗号隔开。不同于 `getHeader($name)` 方法，此方法返回的是由逗号隔开的字符串。

```
$headerValueString = $response->getHeaderLine('Vary');
```

Figure 7: Get single header's values as comma-separated string.

检查响应头

使用 PSR 7 响应对象的 `hasHeader($name)` 方法检查响应头存在与否。

```
if ($response->hasHeader('Vary')) {
    // Do something
}
```

Figure 8: Detect presence of a specific HTTP header.

设置响应头

使用 PSR 7 响应对象的 `withHeader($name, $value)` 方法设置响应头的值。

```
$newResponse = $oldResponse->withHeader('Content-type', 'application/json');
```

Figure 9: Set HTTP header

提示响应对象是不可改的。此方法返回一个响应对象的拷贝（*copy*），它拥有新的值。此方法是破坏性的，它替换了已有的同名响应头现有的值。

追加响应头/Append Header

使用 PSR 7 响应对象的 `withAddedHeader($name, $value)` 方法追加一个响应头的值。

```
$newResponse = $oldResponse->withAddedHeader('Allow', 'PUT');
```

Figure 10: Append HTTP header

提示不同于 `withHeader()` 方法，此方法是追加（*append*）新的值到响应头已有的值中。该响应对象是不可修改的。此方法返回一个已添加新值的该对象的拷贝。

移除响应头

使用 HTTP 响应对象的 `withoutHeader($name)` 方法移除响应头。

```
$newResponse = $oldResponse->withoutHeader('Allow');
```

Figure 11: Remove HTTP header

提示响应对象是不可改的。此方法返回一个带有追加的响应头的值的响应对象的拷贝（*copy*）。

HTTP 响应体

HTTP 响应通常有一个响应体。Slim 提供了一个 PSR 7 响应对象，你可以用它检查或操作可能会有的响应体。

类似 PSR 7 请求对象，PSR 7 响应对象将响应体作为 `\Psr\Http\Message\StreamInterface` 的实例来实现。你可以使用 PSR 7 响应对象的 `getBody()` 方法来获取 HTTP 响应体 `StreamInterface` 的实例。该 `getBody()` 方法完美适用于未知大小或对于可用内容来说太大的输出（outgoing） HTTP 响应。

```
$body = $response->getBody();
```

Figure 12: Get HTTP response body

所得的 `\Psr\Http\Message\StreamInterface` 实例提供以下方法来读取、迭代、写入到潜在的 PHP `资源（resource）` 。

- `getSize()`
- `tell()`
- `eof()`
- `isSeekable()`
- `seek()`
- `rewind()`
- `isWritable()`
- `write($string)`
- `isReadable()`
- `read($length)`
- `getContents()`
- `getMetadata($key = null)`

大多数情况下，你会需要写入到 PSR 7 响应对象。你可以像这样使用 `write()` 方法将内容写入到 `StreamInterface` 实例：

```
$body = $response->getBody();
$body->write('Hello');
```

Figure 13: Write content to the HTTP response body

你同样可以完整新建一个 `StreamInterface` 实例来替换 PSR 7 响应对象的响应体。这玩意特别有用，尤其是你想要从远程地址传输内容到 HTTP 响应中时（例如，文件系统或远程 API）。你可以使用 `withBody(StreamInterface $body)` 方法替换 PSR 7 响应对象的响应体。它的参数必须是 `\Psr\Http\Message\StreamInterface` 的实例。

```
$newStream = new \GuzzleHttp\Psr7\LazyOpenStream('/path/to/file', 'r');
$newResponse = $oldResponse->withBody($newStream);
```

Figure 14: Replace the HTTP response body

提示响应对象不可改变。这个方法返回的是包含新响应体的对象的拷贝（*copy*）。

返回 JSON

Slim 的响应对象拥有一个自定义方法 `withJson($data, $status, $encodingOptions)` 来帮助优化返回 JSON 数据的过程。

其中 `$data` 参数包含你希望返回的 JSON 的数据结构。`$status` 是可选的，并能返回一个自定义的 HTTP 代码。`$encodingOptions` 是可选的，它是与 `json_encode()` 相同的编码选项。

最简单的形式，可以返回带默认的 200 HTTP 状态代码的 JSON 数据。

```
$data = array('name' => 'Bob', 'age' => 40);
$newResponse = $oldResponse->withJson($data);
```

Figure 15: Returning JSON with a 200 HTTP status code.

还可以返回带有自定义 HTTP 状态码的 JSON 数据。

```
$data = array('name' => 'Rob', 'age' => 40);
$newResponse = $oldResponse->withJson($data, 201);
```

Figure 16: Returning JSON with a 201 HTTP status code.

HTTP 响应的 `Content-Type` 自动设置为 `application/json;charset=utf-8`。

如果 JSON 存在数据编码问题，`\RuntimeException($message, $code)` 将抛出异常，包含将 `json_last_error_msg()` 的值作为 `$message` 的值以及将 `json_last_error()` 作为 the `$code` 的值。

提示响应对象是不可改的。此方法返回一个响应对象的拷贝（*copy*），该拷贝带有新的 Content-Type 头。该方法是毁灭性的，它将替换（*replaces*）已存在的 Content-Type 头。当 `withJson()` 被调用，如果传递了 $status ，HTTP 状态码也会被替换。

路由

Slim 框架的路由基于 nikic/fastroute 组件进行构建，它格外地快速稳定。

如何创建路由

你在可以使用 `\Slim\App` 实例的代理（proxy）方法来定义应用程序路由。Slim 框架提供了最流行的HTTP方法。

Get 路由

使用 Slim 应用程序的 `get()` 方法添加一个只处理 `GET` HTTP 请求的路由。它接收两个参数：

1. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
2. 路由回调

```
$app = new \Slim\App();
$app->get('/books/{id}', function ($request, $response, $args) {
  // Show book identified by $args['id']
});
```

POST 路由

使用 Slim 应用程序的 `post()` 方法添加一个只处理 `POST` HTTP 请求的路由。它接收两个参数：

1. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
2. 路由回调

```
$app = new \Slim\App();
$app->post('/books', function ($request, $response, $args) {
  // Create new book
});
```

PUT 路由

使用 Slim 应用程序的 `put()` 方法添加一个只处理 `PUT` HTTP 请求的路由。它接收两个参数：

1. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
2. 路由回调

```
$app = new \Slim\App();
$app->put('/books/{id}', function ($request, $response, $args) {
 // Update book identified by $args['id']
});
```

DELETE 路由

使用 Slim 应用程序的 `delete()` 方法添加一个只处理 `DELETE` HTTP 请求的路由。它接收两个参数：

1. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
2. 路由回调

```
$app = new \Slim\App();
$app->delete('/books/{id}', function ($request, $response, $args) {
 // Delete book identified by $args['id']
});
```

OPTIONS 路由

使用 Slim 应用程序的 `options()` 方法添加一个只处理 `OPTIONS` HTTP 请求的路由。它接收两个参数：

1. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
2. 路由回调

```
$app = new \Slim\App();
$app->options('/books/{id}', function ($request, $response, $args) {
 // Return response headers
});
```

PATCH 路由

使用 Slim 应用程序的 `patch()` 方法添加一个只处理 `PATCH` HTTP 请求的路由。它接收两个参数：

1. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
2. 路由回调

```
$app = new \Slim\App();
$app->patch('/books/{id}', function ($request, $response, $args) {
 // Apply changes to book identified by $args['id']
});
```

任意路由

使用 Slim 应用程序的 `any()` 方法添加一个可以处理处理所有 HTTP 请求的路由。它接收两个参数：

1. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
2. 路由回调

```
$app = new \Slim\App();
$app->any('/books/[{id}]', function ($request, $response, $args) {
 // Apply changes to books or book identified by $args['id'] if specified.
 // To check which method is used: $request->getMethod();
});
```

记住，第二个参数是一个回调。你需要指定一个类（Class，需要一个 `__invoke()` 实现方法。）来替换闭包（Closure）。接着，你可以映射到其他位置：

```
$app->any('/user', 'MyRestfulController');
```

自定义路由

使用 Slim 应用程序的 `map()` 方法来添加一个可以处理多个 HTTP 请求方法的路由。它接收三个参数：

1. HTTP 方法的数组
2. 路由模式（带有可选的命名占位符）/The route pattern (with optional named placeholders)
3. 路由回调

```
$app = new \Slim\App();
$app->map(['GET', 'POST'], '/books', function ($request, $response, $args) {
 // Create new book or list all books
});
```

路由回调

上述的每个路由方法都接收一个回调例程作为其最后一个参数。这个参数可以是任意 PHP 调用（callable），并且它默认接受三个参数。

请求/Request

第一个参数是一个 `Psr\Http\Message\ServerRequestInterface` 对象，表示当前的 HTTP 请求。

响应/Response

第二个参数是一个 `Psr\Http\Message\ResponseInterface` 对象，表示当前的 HTTP 响应。

参数数组/Arguments

第三个参数是一个关联数组，包含包含当前路由的命名占位符。

将内容写入响应

有两种方式可以将内容写入 HTTP 响应。第一钟，可以使用 `echo()` 简单地从路由回调中输出内容。其内容将会呗追加到当前 HTTP 请求对象中。第二种，你可以返回一个 `Psr\Http\Message\ResponseInterface` 对象。

闭包绑定/Closure binding

如果你使用一个`闭包（Closure）`实例作为路由回调，闭包的状态受`Container`实例约束。这意味着你将通过 `$this` 关键字访问闭包内部的 DI 容器实例：

```
$app = new \Slim\App();
$app->get('/hello/{name}', function ($request, $response, $args) {
 // Use app HTTP cookie service
 $this->get('cookies')->set('name', [
 'name' => $args['name'],
 'expires' => '7 days'
 ]);
});
```

路由策略

路由回调签名由路由策略决定。默认地，Slim 寄望路由回调来接收请求、响应和由路由占位符参数组成的数组。这称为请求响应策略（RequestResponse strategy）。然而，你可以通过使用另一个不同策略来改变这种寄望。例如，Slim 提供了一个另类的策略，叫做 RequestResponseArgs ,它接收请求和响应，加上由每一个路由占位符组成的单独参数。这里的例子展示了如何使用这个另类的策列；轻松替代了默认`\Slim\Container`提供的 `foundHandler` 依赖：

```
$c = new \Slim\Container();
$c['foundHandler'] = function() {
 return new \Slim\Handlers\Strategies\RequestResponseArgs();
};

$app = new \Slim\App($c);
$app->get('/hello/{name}', function ($request, $response, $name) {
 return $response->write($name);
});
```

通过实现 `\Slim\Interfaces\InvocationStrategyInterface` 你可以提供一个你自己的路由策略。

路由占位符

上诉的每个路由方法都会收到一个 URL 模式（URL pattern），它将与当前 HTTP 请求的 URI 相匹配。路由模式将使用命名占位符（named placeholder）来动态匹配 HTTP 请求的 URI 片段。

格式

路由模式占位符起始与一个 `{` , 然后是占位符名称, 最后以 `}` 结束。这是一个名为 `name` 的占位符例子：

```
$app = new \Slim\App();
$app->get('/hello/{name}', function ($request, $response, $args) {
 echo "Hello, " . $args['name'];
});
```

可选的分段 / Optional segments

使一个片段可选，只需用将其放在方括号中：

```
$app->get('/users[/{id}]', function ($request, $response, $args) {
    // reponds to both `/users` and `/users/123`
    // but not to `/users/`
});
```

支持多个可选参数嵌套：

```
$app->get('/news[/{year}[/{month}]]', function ($request, $response, $args) {
    // reponds to `/news`, `/news/2016` and `/news/2016/03`
});
```

对于数目不确定的可选参数，可以这样做：

```
$app->get('/news[/{params:.*}]', function ($request, $response, $args) {
    $params = explode('/', $request->getAttribute('params'));

    // $params is an array of all the optional segments
});
```

在这个例子中，`/news/2016/03/20` 的 URI 将使得 `$params` 数组包含三个元素：`['2016', '03', '20']`.

正则表达式匹配

默认地，占位符被写在 `{}` 之中，并可以接收任意值。然而，占位符同样可以要求 HTTP 请求 URI 匹配特定的正则表达式。如果当前的HTTP 请求URI 不能与占位符的正则表达式匹配，路由路由将不会启动。下面这个示例占位符要求 `id` 必须是个数字：

```
$app = new \Slim\App();
$app->get('/users/{id:[0-9]+}', function ($request, $response, $args) {
 // Find user identified by $args['id']
});
```

路由名称

应用程序的路由可以被指定一个名称。这非常有用，如果你想要使用路由的 `pathFor()` 方法以编程的形式生成一个特定路由的 URL 。上述的每个路由方法都会返回一个 `\Slim\Route` 对象，这个对象带来了 `setName()` 方法。

```
$app = new \Slim\App();
$app->get('/hello/{name}', function ($request, $response, $args) {
 echo "Hello, " . $args['name'];
})->setName('hello');
```

使用应用程序路由的 `pathFor()` 方法，为已命名的路由生成一个 URL 。

```
echo $app->router->pathFor('hello', [
 'name' => 'Josh'
]);
// Outputs "/hello/Josh"
```

路由的 `pathFor()` 方法接收两个参数：

1. 路由名称
2. 由路由模式占位符及替换值组成的关联数组。

路由组

为了帮助将路由整理成条理分明的路由组，`\Slim\App` 还提供了一个 `group()` 方法。 每个路由组的路由模式预置于路由或路由组中的路由组，路由组模式的任何占位符参数最终使得嵌套的路由都是可用的：

```
$app = new \Slim\App();
$app->group('/users/{id:[0-9]+}', function () {
 $this->map(['GET', 'DELETE', 'PATCH', 'PUT'], '', function ($request, $response, $args)
 // Find, delete, patch or replace user identified by $args['id']
 })->setName('user');
 $this->get('/reset-password', function ($request, $response, $args) {
 // Route for /users/{id:[0-9]+}/reset-password
 // Reset the password for user identified by $args['id']
 })->setName('user-password-reset');
});
```

记住，在路由组闭包的内部，使用 `$this` 替代 `$app` 。Slim 将闭包绑定到应用程序实例，就像路由回调的情况那样。

路由中间件

你可以将中间件与任意路由或路由组相连接。 了解更多.

容器识别 / Container Resolution

你不必限于为路由定义函数。在 Slim 中，有一些不同的方式来定义你的路由行为函数。

除了函数外，你还可以使用: - 可调用的类 - `Class:method`

这个函数由 Slim 的 Callable 解角器（Resolver） 类提供支持。它将字符串入口（entry）转变为函数调用。例如：

```
$app->get('/home', '\HomeController:home');
```

在上面这段代码中，我们定义了一个 `/home` 路由，并告诉 Slim 执行 `\HomeController` 类中的 `home()` 方法。

Slim 首先在容器中寻找 `\HomeController` 的入口，如果找到了，它将使用该实例。否则，它将它的构造函数，并将容器作为第一个参数。一旦这个类的实例创建了，它将使用你已定义的策略（Strategy）去调用指定的方法。

作为另一种办法，你可以使用一个可调用的（invokable）类，比如：

```
class MyAction {
   protected $ci;
   //Constructor
   public function __construct(ContainerInterface $ci) {
       $this->ci = $ci;
   }

   public function __invoke($request, $response, $args) {
        //your code
        //to access items in the container... $this->ci->get('');
   }
}
```

你可以这样使用这个累：

```
$app->get('/home', '\MyAction');
```

在更为传统的 MVC 方法中，你构建包含许多行为（actions）的控制器来替代只能处理一个行为的可调用类。

```
class MyController {
   protected $ci;
   //Constructor
   public function __construct(ContainerInterface $ci) {
       $this->ci = $ci;
   }

   public function method1($request, $response, $args) {
        //your code
        //to access items in the container... $this->ci->get('');
   }

   public function method2($request, $response, $args) {
        //your code
        //to access items in the container... $this->ci->get('');
   }

   public function method3($request, $response, $args) {
        //your code
        //to access items in the container... $this->ci->get('');
   }
}
```

你可以这样使用控制器方法：

```
$app->get('/method1', '\MyController:method1');
$app->get('/method2', '\MyController:method2');
$app->get('/method3', '\MyController:method3');
```

错误处理

500 系统错误处理器

Edit This Page

出问题了！你并不能预知错误，但你可以预料到。每个 Slime 框架应用程序都有一个错误处理器用于接收所有未抓取的 PHP 异常。这个错误处理器同样能接收当前 HTTP 的请求对象和响应对象。这个错误处理器必须预备并返回一个适当的响应对象，这个对象会被返回到 HTTP 客户端。

默认的错误处理器

默认的错误处理器非常基础。它将响应状态编码设置为 `500`，并将响应内容类型设置为 `text/html`，并将一个通用的错误信息加入到响应体中。

这个对于生产应用大概不太合适。强烈建议你实现专用于你的 Slim 应用程序的错误处理器。

默认的错误处理程序还可以包括详细的错误诊断信息。要启用这个功能你需要将 `displayErrorDetails` 设置为 true :

```
$configuration = [
 'settings' => [
 'displayErrorDetails' => true,
 ],
];
$c = new \Slim\Container($configuration);
$app = new \Slim\App($c);
```

自定义错误处理器

Slim 框架应用程序的错误处理器是一种 Pimple 服务。你可以通过应用程序容器对自定义 Pimple factory 方法进行定义，来创建自定义的错误处理器取代默认的。

有两种注入处理器的方法：

Pre App

```
$c = new \Slim\Container();
$c['errorHandler'] = function ($c) {
 return function ($request, $response, $exception) use ($c) {
 return $c['response']->withStatus(500)
 ->withHeader('Content-Type', 'text/html')
 ->write('Something went wrong!');
 };
};
$app = new \Slim\App($c);
```

Post App

```
$app = new \Slim\App();
$c = $app->getContainer();
$c['errorHandler'] = function ($c) {
 return function ($request, $response, $exception) use ($c) {
 return $c['response']->withStatus(500)
 ->withHeader('Content-Type', 'text/html')
 ->write('Something went wrong!');
 };
};
```

在这个例子中，我们定义了一个新的 `errorHandler` factory ，它将返回一个 callable 。返回的 callable 接收三个参数：

1. 一个 `\Psr\Http\Message\ServerRequestInterface` 实例
2. 一个 `\Psr\Http\Message\ResponseInterface` 实例
3. 一个 `\Exception` 实例

这个 callable 必须 返回一个新的 `\Psr\Http\Message\ResponseInterface` 实例，对于给定的异常也是如此。

务必注意：下面这三个类型的异常不会被自定义的 `errorHandler` 处理：

- `Slim\Exception\MethodNotAllowedException`: 这个可以用自定义的 `notAllowedHandler` 来处理。
- `Slim\Exception\NotFoundException`: 这个可以用自定义的 `notFoundHandler` 来处理。
- `Slim\Exception\SlimException`: 这种类型的异常是 Slim 内置的，它的处理不能被覆写。

禁用

要想彻底地禁用 Slim 的错误处理器，只需从容器中移除错误处理器即可：

```
unset($app->getContainer()['errorHandler']);
```

现在，你需要负责处理所有异常了，因为 Slim 已经不再处理它们。

404 Not Found 处理器

Edit This Page

如果你的 Slim 应用程序没有可以匹配到当前 HTTP 请求 URI 的路由，程序会调用 Not Found 处理器，并返回一个 `HTTP/1.1 404 Not Found` 响应到 HTTP 客户端。

默认的 Not Found 处理器

每个 Slim 框架应用程序都有一个默认的 Not Found 处理器。这个处理器将响应状态设置为 `404` ，并将内容类型设置为 `text/html` ，它将写入一个简单的异常信息到响应体。

自定义 Not Found 处理器

Slim 框架应用程序的 Not Found 处理器是一个 Pimple 服务。 你可以通过应用程序容器对自定义 Pimple factory 方法进行定义，来创建自定义的 Not Found 处理器取代默认的。

```
$c = new \Slim\Container(); //Create Your container

//Override the default Not Found Handler
$c['notFoundHandler'] = function ($c) {
 return function ($request, $response) use ($c) {
 return $c['response']
 ->withStatus(404)
 ->withHeader('Content-Type', 'text/html')
 ->write('Page not found');
 };
};

//Create Slim
$app = new \Slim\App($c);

//... Your code
```

在这个例子中，我们定义了一个新的 `notFoundHandler` factory ，它将返回一个 callable 。返回的 callable 接收2个参数：

1. 一个 `\Psr\Http\Message\ServerRequestInterface` 实例
2. 一个 `\Psr\Http\Message\ResponseInterface` 实例

这个 callable 必须 返回一个恰当的 `\Psr\Http\Message\ResponseInterface` 实例。

405 Not Allowed 处理器

Edit This Page

如果你的 Slim 框架应用程序有一个路由匹配到了当前的 HTTP 请求 URI 而非 **HTTP** 请求方法，程序将调用 Not Allowed 处理器并返回一个 `HTTP/1.1 405 Not Allowed` 响应到 HTTP 客户端。

默认的 Not Allowed 处理器

每个 Slim 框架应用程序都有一个默认的 Not Allowed 处理器。该处理器将 HTTP 响应状态设置为 `405` ，将内容类型设置为 `text/html` ，它还会添加一个包含由逗号分隔的已被允许访问的 HTTP 方法组成的列表的 `Allowed:` HTTP 头它还会在 HTTP 响应体中写入一个简单的注释。

自定义 Not Allowed 处理器

Slim 框架应用程序的 Not Allowed 处理器是一个 Pimple 服务。你可以通过应用程序容器器对自定义 Pimple factory 方法进行定义，来创建自定义的 Not Allowed 处理器取代默认的

```
// Create Slim
$app = new \Slim\App();
// get the app's di-container
$c = $app->getContainer();
$c['notAllowedHandler'] = function ($c) {
 return function ($request, $response, $methods) use ($c) {
 return $c['response']
 ->withStatus(405)
 ->withHeader('Allow', implode(', ', $methods))
 ->withHeader('Content-type', 'text/html')
 ->write('Method must be one of: ' . implode(', ', $methods));
};
```

> 注意 Check out Not Found docs for pre-slim creation method using a new instance of `\Slim\Container`

在这个例子中，我们定义了一个新的 `notAllowedHandler` factory ，它将返回一个 callable 。返回的 callable 接收两个参数：

1. 一个 `\Psr\Http\Message\ServerRequestInterface` 实例
2. 一个 `\Psr\Http\Message\ResponseInterface` 实例
3. 一个由已允许访问的 HTTP 方法名组成的数组

这个 callable 必须 返回一个恰当的 `\Psr\Http\Message\ResponseInterface` 实例。

烹饪书

以 / 结尾的路由模式

Edit This Page

Slim 处理带有斜线结尾的 URL 和不带斜线的 URL 的方式不同。意思就是 `/user` 和 `/user/` 不是一回事，它们可以有不同的回调。

如果你想通过重定向让所有以 `/` 结尾的 URL 和不以 `/` 结尾的 URL 相等，你可以添加这个中间件：

```
use Psr\Http\Message\RequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

$app->add(function (Request $request, Response $response, callable $next) {
 $uri = $request->getUri();
 $path = $uri->getPath();
 if ($path != '/' && substr($path, -1) == '/') {
 // permanently redirect paths with a trailing slash
 // to their non-trailing counterpart
 $uri = $uri->withPath(substr($path, 0, -1));
 return $response->withRedirect((string)$uri, 301);
 }

 return $next($request, $response);
});
```

或者，也可以使用 oscarotero/psr7-middlewares' TrailingSlash 中间件强制为所有 URL 添加结尾的斜线：

```
use Psr7Middlewares\Middleware\TrailingSlash;

$app->add(new TrailingSlash(true)); // true 则添加结尾斜线 (false 移除斜线)
```

检索 IP 地址

检索客户端当前 IP 地址的最佳方式，是利用使用了类似 rka-ip-address-middleware 这种组件的中间件。

这个组件可以通过 composer 来安装：

```
composer require akrabat/rka-ip-address-middleware
```

要使用这个组件，需要使用 `App` 注册中间件，这里提供了一个可信赖的代理列表（e.g. varnish 服务器）, 如果你再使用它们：

```
$checkProxyHeaders = true;
$trustedProxies = ['10.0.0.1', '10.0.0.2'];
$app->add(new RKA\Middleware\IpAddress($checkProxyHeaders, $trustedProxies));

$app->get('/', function ($request, $response, $args) {
    $ipAddress = $request->getAttribute('ip_address');

    return $response;
});
```

这个中间件把客户端 IP 地址存储在一个 HTTP 请求属性中，所以需要通过 `$request->getAttribute('ip_address')` 来访问。

检索当前路由

如果你需要在应用程序中获取当前的路由，你所需要做但就是，调用 HTTP 请求类的带有 `'route'` 参数的 `getAttribute` 方法，它将返回当前的路由，这个路由是 `Slim\Route` 类的实例。class.

可以使用 `getName()` 获取路由的名称，或者使用 `getMethods()` 获取此路由支持的方法, etc。

Note: 如果你需要在 app 中间件中访问路由，必须在配置中将 `'determineRouteBeforeAppMiddleware'` 设置为 true。否则，`getAttribute('route')` 将会返回 null。该路由在路由中间件中永远可用。

Example:

```
use Slim\Http\Request;
use Slim\Http\Response;
use Slim\App;

$app = new App([
    'settings' => [
        // Only set this if you need access to route within middleware
        'determineRouteBeforeAppMiddleware' => true
    ]
])

// routes...
$app->add(function (Request $request, Response $response, callable $next) {
    $route = $request->getAttribute('route');
    $name = $route->getName();
    $groups = $route->getGroups();
    $methods = $route->getMethods();
    $arguments = $route->getArguments();

    // do something with that information

    return $next($request, $response);
});
```

在 Slim 中使用 Eloquent

你可以使用 Eloquent 这种数据库 ORM 将你的 Slim 应用程序连接到数据库。

为你的应用程序添加 Eloquent

```
composer require illuminate/database "~5.1"
```

Figure 1: Add Eloquent to your application.

配置 Eloquent

在 Slim 的 setting 数组中添加数据库设置项。

```
<?php
return [
    'settings' => [
        // Slim Settings
        'determineRouteBeforeAppMiddleware' => false,
        'displayErrorDetails' => true,
        'db' => [
            'driver' => 'mysql',
            'host' => 'localhost',
            'database' => 'database',
            'username' => 'user',
            'password' => 'password',
            'charset'   => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix'    => '',
        ]
    ],
];
```

Figure 2: Settings array.

在 `dependencies.php` 中，或者其他任意位置添加你的 Service Factories:

```
// Service factory for the ORM
$container['db'] = function ($container) {
    $capsule = new \Illuminate\Database\Capsule\Manager;
    $capsule->addConnection($container['settings']['db']);

    $capsule->setAsGlobal();
    $capsule->bootEloquent();

    return $capsule;
};
```

Figure 3: Configure Eloquent.

传递数据表实例到控制器

```
$container[App\WidgetController::class] = function ($c) {
    $view = $c->get('view');
    $logger = $c->get('logger')
    $table = $c->get('db')->table('table_name');
    return new \App\WidgetController($view, $logger, $table);
};
```

Figure 4: Pass table object into a controller.

从控制器中查询数据表

```
<?php

namespace App;

use Slim\Views\Twig;
use Psr\Log\LoggerInterface;
use Illuminate\Database\Query\Builder;
use Psr\Http\Message\ServerRequestInterface as Request;
use Psr\Http\Message\ResponseInterface as Response;

class WidgetController
{
    private $view;
    private $logger;
    protected $table;

    public function __construct(
        Twig $view,
        LoggerInterface $logger,
        Builder $table
    ) {
        $this->view = $view;
        $this->logger = $logger;
        $this->table = $table;
    }

    public function __invoke(Request $request, Response $response, $args)
    {
        $widgets = $this->table->get();

        $this->view->render($response, 'app/index.twig', [
            'widgets' => $widgets
        ]);

        return $response;
    }
}
```

Figure 5: Sample controller querying the table.

使用 where 查询数据表

```
...
$records = $this->table->where('name', 'like', '%foo%')->get();
...
```

Figure 6: Query searching for names matching foo.

通过 id 查询数据表

```
...
$record = $this->table->find(1);
...
```

Figure 7: Selecting a row based on id.

了解更多

Eloquent 文档

附加组件

模板

Slim 没有传统 MVC 框架的视图（view）层。相反，Slim 的“视图”就是 HTTP 响应。Slim 应用程序的每个路由都为准备和返回恰当的 PSER 7 响应对象负责。

> Slim's "view" is the HTTP response.

话虽如此，但 Slim 项目提供了 Twig-View 和 PHP-View 组件帮助你将模版渲染为 PSR7 响应对象。

The slim/twig-view 组件

Twig-View PHP 组件帮助你渲染应用程序中的 Twig 模版。这个组件可以在 Packageist 上找到。可以像这样使用 composer 轻易地安装：

```
composer require slim/twig-view
```

Figure 1: Install slim/twig-view component.

下一步，你需要在 Slim 应用容器中将此组将注册为服务，像这样：

```
<?php
// Create app $app = new \Slim\App();

// Get container $container = $app->getContainer();

// Register component on container $container['view'] = function ($container) {
    $view = new \Slim\Views\Twig('path/to/templates', [
        'cache' => 'path/to/cache'
    ]);
    $view->addExtension(new \Slim\Views\TwigExtension(
        $container['router'],
        $container['request']->getUri()
    ));

    return $view;
};
```

Figure 2: Register slim/twig-view component with container.

记住：“cache” 可以设置为 false 禁用它，‘auto_reload’ 选项也是如此，这在开发环境中很有用。了解更多信息，查阅： Twig environment options

现在你可以使用应用程序内部的 `slim/twig-view` 组件服务并将其写入到 PSR 7 响应对象中，像这样：

```
// Render Twig template in route
$app->get('/hello/{name}', function ($request, $response, $args) {
    return $this->view->render($response, 'profile.html', [
        'name' => $args['name']
    ]);
})->setName('profile');

// Run app
$app->run();
```

Figure 3: Render template with slim/twig-view container service.

在这个例子中，在路由回调中被调用的 `$this->view`，是容器服务返回的 `\Slim\Views\Twig` 实例的一个参考（reference）。`\Slim\Views\Twig` 实例的 `render()` 方法接收一个 PSR7 响应对象作为它的第一个参数，Twig 模版路径作为它的第二个参数，模板变量的数组作为它的最后一个参数。这个 `render()` 方法返回一个新的 PSR7 响应对象，它的响应体是由 Twig 模版渲染的。

path_for() 方法

`slim/twig-view` 组件为 Twig 模版暴露了一个自定义 `path_for()` 函数。你可以使用这个函数生成完整的指向应用程序中任意已命名路由的 URL。`path_for()` 函数接收两个参数：

1. 路由名称
2. 路由占位符和替换值的散列（hash）

第二个参数的关键字须与已选择的路由的模式占位符一致。这是一个示例的 Twig 模版，它描述了上面的示例 Slim 应用程序中的 “profile” 路由的链接 URL。

```
{% extends "layout.html" %}

{% block body %}
<h1>User List</h1>
<ul>
    <li><a href="{{ path_for('profile', { 'name': 'josh' }) }}">Josh</a></li>
</ul>
{% endblock %}
```

slim/php-view 组件

PHP-View PHP 组件帮助你渲染 PHP 模版。该组件可以在 Packagist 上找到，像这样使用 Composer 安装：

```
composer require slim/php-view
```

Figure 4: Install slim/php-view component.

在 Slim App 的容器中，将此组件注册为服务，这么做：

```
<?php
// Create app $app = new \Slim\App();

// Get container $container = $app->getContainer();

// Register component on container $container['view'] = function ($container) {
    return new \Slim\Views\PhpRenderer('path/to/templates/with/trailing/slash/');
};
```

Figure 5: Register slim/php-view component with container.

使用 view 组件渲染 PHP 视图：

```
// Render Twig template in route
$app->get('/hello/{name}', function ($request, $response, $args) {
    return $this->view->render($response, 'profile.html', [
        'name' => $args['name']
    ]);
})->setName('profile');

// Run app
$app->run();
```

Figure 6: Render template with slim/twig-view container service.

其他模版系统

并不限于使用 `Twig-View` 和 `PHP-View` 组件。你可以使用任意 PHP 模版系统，只要它能渲染你的模版，并最终输出到 PSR7 响应对象的 body 中。

HTTP 缓存

Edit This Page

Slim 3 使用 slimphp/Slim-HttpCache 这款独立的 PHP 组件作为可选的 HTTP 缓存工具。可以使用这个组件创建并返回包含 `Cache` , `Expires` , `ETag` , 和 `Last-Modified` 头的 HTTP 响应，以控制何时以及如何使用客户端缓存保留应用程序的输出。

安装

在你的项目的根目录下执行这个bash命令：

```
composer require slim/http-cache
```

用法

这个 `slimphp/Slim-HttpCache` 组件包含一个服务提供程序和一个应用程序中间件。你必须在你的应用程序中添加这两个玩意，像这样：

```
// Register service provider with the container
$container = new \Slim\Container;
$container['cache'] = function () {
 return new \Slim\HttpCache\CacheProvider();
};

// Add middleware to the application
$app = new \Slim\App($container);
$app->add(new \Slim\HttpCache\Cache('public', 86400));

// Create your application routes...

// Run application
$app->run();
```

ETag

使用服务提供程序的 `withEtag()` 方法创建一个带有指定 `ETag` 头的响应对象。这个方法接收一个 PSR7 响应对象，而且它返回一个带有新 HTTP 头的 PSR7 响应的拷贝。

```
$app->get('/foo', function ($req, $res, $args) {
 $resWithEtag = $this->cache->withEtag($res, 'abc');

 return $resWithEtag;
});
```

Expires

使用服务提供程序的 `withExpires()` 方法创建一个带有指定 `Expires` 头的响应对象。这个方法接收一个 PSR7 响应对象，而且它返回一个带有新的 HTTP 头的 PSR7 响应的拷贝。

```
$app->get('/bar',function ($req, $res, $args) {
 $resWithExpires = $this->cache->withExpires($res, time() + 3600);

 return $resWithExpires;
});
```

Last-Modified

使用服务提供程序的`withLastModified()` 方法创建一个带有指定 `Last-Modified` 头的响应对象。这个方法接收一个 PSR7 响应对象，而且它返回一个带有新 HTTP 头的 PSR7 响应的拷贝。

```
//Example route with LastModified
$app->get('/foobar',function ($req, $res, $args) {
 $resWithLastMod = $this->cache->withLastModified($res, time() - 3600);

 return $resWithLastMod;
});
```

CSRF 保护

Edit This Page

Slim 3 使用独立可选的 PHP 组件 slimphp/Slim-Csrf 来保护应用程序免遭 CSRF（跨站请求伪造）。本组件为每个请求生成一个唯一 token ，验证来源于客户端 HTML 表单产生的 POST 请求。

安装

在你的项目的根目录下执行这个bash命令：

```
composer require slim/csrf
```

用法

这个 `slimphp/Slim-Csrf` 组件包含一个应用程序中间件。像这样将它添加到你的应用程序中：

```
// Add middleware to the application
$app = new \Slim\App;
$app->add(new \Slim\Csrf\Guard);

// Create your application routes...

// Run application
$app->run();
```

提取 CSRF token 的名称和值

最新的 CSRF token 作为 PSR7 请求对象的属性，其名称和值是可获取的。每个请求的CSRF token 都是唯一的。你可以像这样提取当前的 CSRF token 的名称和值：

```
$app->get('/foo', function ($req, $res, $args) {
 // Fetch CSRF token name and value
 $name = $req->getAttribute('csrf_name');
 $value = $req->getAttribute('csrf_value');

 // TODO: Render template with HTML form and CSRF token hidden field
});
```

你应该将 CSRF token 的名称和值船体给模板，这样它们就能和 HTML 表单 POST 请求一起被提交。它们经常被存储为 HTML 表单的一个隐藏字段。

Flash Messages

Coming soon.